Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for reuse

Design patterns for reuse, part 2

Michael Hilton  Bogdan Vasilescu
Administrivia

- HW 3 deadline Tue, Feb 12 instead of Thu, Feb 7

- Optional: UML and Patterns, Ch. 17 (Design principles)

- Optional: Effective Java,
  - Item 49 (Check parameters for validity)
  - Item 54 (Return empty collections or arrays, not nulls)
  - Item 69 (Use exceptions only for exceptional conditions)

- Midterm 1 on Thu, Feb 14
  - Review meeting: Wed, Feb 13, 6-8pm, Scaife Hall 125
Key concepts from last Tuesday
UML you should know

• Interfaces vs. classes
• Fields vs. methods
• Relationships:
  – "extends" (inheritance)
  – "implements" (realization)
  – "has a" (aggregation)
  – non-specific association
• Visibility:  + (public)  - (private)  # (protected)
• Basic best practices...
Design patterns

• Carpentry:
  – "Is a dovetail joint or a miter joint better here?"

• Software Engineering:
  – "Is a strategy pattern or a template method better here?"
Elements of a design pattern

- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences
Strategy pattern

- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes:
    - Code can be harder to understand
    - Lots of overhead if the strategies are simple
Template method pattern

- **Problem**: An algorithm consists of customizable parts and invariant parts
- **Solution**: Implement the invariant parts of the algorithm in an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations
- **Consequences**
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations
  - Inverted (Hollywood-style) control for customization
Today

• More design patterns for reuse
  – Iterator pattern
  – Decorator pattern

• Design goals and design principles
Traversing a collection

- Old-school Java for loop for ordered types
  ```java
  List<String> arguments = ...;
  for (int i = 0; i < arguments.size(); i++) {
      System.out.println(arguments.get(i));
  }
  ```

- Modern standard Java for-each loop
  ```java
  List<String> arguments = ...;
  for (String s : arguments) {
      System.out.println(s);
  }
  ```

Works for every implementation of `Iterable`:
```java
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```
The Iterator interface

```java
public interface java.util.Iterator<E> {
    boolean hasNext();
    E next();
    void remove();  // removes previous returned item
}                 // from the underlying collection
```

- To use explicitly, e.g.:
  ```java
  List<String> arguments = ...;
  for (Iterator<String> it = arguments.iterator();
    it.hasNext(); ) {
    String s = it.next();
    System.out.println(s);
  }
  ```
Getting an Iterator

```
public interface Collection<E> extends Iterable<E> {
    boolean add(E e);
    boolean addAll(Collection<? extends E> c);
    boolean remove(Object e);
    boolean removeAll(Collection<?> c);
    boolean contains(Object e);
    boolean containsAll(Collection<?> c);
    void clear();
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    ...
}
```

Defines an interface for creating an Iterator, but allows Collection implementation to decide which Iterator to create.
An Iterator implementation for Pairs

```java
public class Pair<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
}
```

```java
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { ... }
```
An Iterator implementation for Pairs

```java
public class Pair<E> implements Iterable<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
    public Iterator<E> iterator() {
        return new PairIterator();
    }
}
```

```java
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { ... }
```
An Iterator implementation for Pairs

```java
public class Pair<E> implements Iterable<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
    public Iterator<E> iterator() {
        return new PairIterator();
    }
    private class PairIterator implements Iterator<E> {
        public boolean hasNext() {
        }
        public E next() {
        }
        public void remove() {
        }
    }
    public void void remove() {
    }
}
Pair<String> pair = new Pair<String>("foo", "bar");
for (String s : pair) { ... }
```
An Iterator implementation for Pairs

```java
public class Pair<E> implements Iterable<E> {
    private final E first, second;
    public Pair(E f, E s) { first = f; second = s; }
    public Iterator<E> iterator() {
        return new PairIterator();
    }
    private class PairIterator implements Iterator<E> {
        private boolean seenFirst = false, seenSecond = false;
        public boolean hasNext() { return !seenSecond; }
        public E next() {
            if (!seenFirst) { seenFirst = true; return first; }
            if (!seenSecond) { seenSecond = true; return second; }
            throw new NoSuchElementException();
        }
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
    Pair<String> pair = new Pair<String>("foo", "bar");
    for (String s : pair) { ... }
}
```
Iterator design pattern

• Problem: Clients need uniform strategy to access all elements in a container, independent of the container type
  – Order is unspecified, but access every element once
• Solution: A strategy pattern for iteration
• Consequences:
  – Hides internal implementation of underlying container
  – Easy to change container type
  – Facilitates communication between parts of the program
Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable...
- ...but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`
Using a `java.util.Iterator<E>`: A warning

- The default Collections implementations are mutable...
- ...but their `Iterator` implementations assume the collection does not change while the `Iterator` is being used
  - You will get a `ConcurrentModificationException`
  - If you simply want to remove an item:
    ```java
    List<String> arguments = ...;
    for (Iterator<String> it = arguments.iterator();
         it.hasNext(); ) {
       String s = it.next();
       if (s.equals("Bogdan"))
         arguments.remove("Bogdan"); // runtime error
    }
    ```
Using a java.util.Iterator<E>: A warning

- The default Collections implementations are mutable...
- ...but their Iterator implementations assume the collection does not change while the Iterator is being used
  - You will get a ConcurrentModificationException
  - If you simply want to remove an item:
    ```java
    List<String> arguments = ...;
    for (Iterator<String> it = arguments.iterator();
        it.hasNext(); ) {
        String s = it.next();
        if (s.equals("Bogdan"))
            it.remove();
    }
    ```
Today

• More design patterns for reuse
  – Iterator pattern
  – Decorator pattern
• Design goals and design principles
Limitations of inheritance

• Suppose you want various extensions of a Stack data structure...
  – UndoStack: A stack that lets you undo previous push or pop operations
  – SecureStack: A stack that requires a password
  – SynchronizedStack: A stack that serializes concurrent accesses
Limitations of inheritance

- Suppose you want various extensions of a Stack data structure...
  - UndoStack: A stack that lets you undo previous push or pop operations
  - SecureStack: A stack that requires a password
  - SynchronizedStack: A stack that serializes concurrent accesses
  - SecureUndoStack: A stack that requires a password, and also lets you undo previous operations
  - SynchronizedUndoStack: A stack that serializes concurrent accesses, and also lets you undo previous operations
  - SecureSynchronizedStack: ...
  - SecureSynchronizedUndoStack: ...

Goal: arbitrarily composable extensions
Limitations of inheritance

Extensions not combinable

Middle extension not optional
Workarounds?

- Combining inheritance hierarchies
  - Combinatorical explosion
  - Massive code replication

Multiple inheritance
- Diamond problem
The decorator design pattern

• Problem: You need arbitrary or dynamically composable extensions to individual objects.

• Solution: Implement a common interface as the object you are extending, add functionality, but delegate primary responsibility to an underlying object.

• Consequences:
  – More flexible than static inheritance
  – Customizable, cohesive extensions
  – Breaks object identity, self-references
Decorators use both subtyping and delegation

```java
public class LoggingList<E> implements List<E> {
    private final List<E> list;
    public LoggingList<E>(List<E> list) { this.list = list; }
    public boolean add(E e) {
        System.out.println("Adding " + e);
        return list.add(e);
    }
    public E remove(int index) {
        System.out.println("Removing at " + index);
        return list.remove(index);
    }
    ...
}
```
A Decorator for our Stack example
The AbstractStackDecorator forwarding class

```java
public abstract class StackDecorator implements IStack {
    private final IStack stack;
    public StackDecorator(IStack stack) {
        this.stack = stack;
    }
    public void push(Item e) {
        stack.push(e);
    }
    public Item pop() {
        return stack.pop();
    }
    ...
}
```
The concrete decorator classes

```java
public class UndoStack extends StackDecorator
    implements IStack {
    private final UndoLog log = new UndoLog();
    public UndoStack(IStack stack) { super(stack); }
    public void push(Item e) {
        log.append(UndoLog.PUSH, e);
        super.push(e);
    }
    ...
}
```
Using the decorator classes

- To construct a plain stack:
  
  ```java
  Stack s = new Stack();
  ```

- To construct an undo stack:
Using the decorator classes

- To construct a plain stack:
  ```java
  Stack s = new Stack();
  ```

- To construct an undo stack:
  ```java
  UndoStack s = new UndoStack(new Stack());
  ```
Using the decorator classes

• To construct a plain stack:
  Stack s = new Stack();

• To construct an undo stack:
  UndoStack s = new UndoStack(new Stack());

• To construct a secure synchronized undo stack:
Using the decorator classes

- To construct a plain stack:
  \[
  \text{Stack } s = \text{new Stack();}
  \]

- To construct an undo stack:
  \[
  \text{UndoStack } s = \text{new UndoStack(new Stack());}
  \]

- To construct a secure synchronized undo stack:
  \[
  \text{SecureStack } s = \text{new SecureStack(new SynchronizedStack(new UndoStack(new Stack())));}\]
Decorators from java.util.Collections

• Turn a mutable list into an immutable list:
  static List<T> unmodifiableList(List<T> lst);
  static Set<T> unmodifiableSet(Set<T> set);
  static Map<K,V> unmodifiableMap(Map<K,V> map);

• Similar for synchronization:
  static List<T> synchronizedList(List<T> lst);
  static Set<T> synchronizedSet(Set<T> set);
  static Map<K,V> synchronizedMap(Map<K,V> map);
The UnmodifiableCollection (simplified excerpt)

```java
public static <T> Collection<T> unmodifiableCollection(Collection<T> c) {
    return new UnmodifiableCollection<>(c);
}

class UnmodifiableCollection<E> implements Collection<E>, Serializable {
    final Collection<E> c;
    UnmodifiableCollection(Collection<> c) {this.c = c; }
    public int size() {return c.size();}
    public boolean isEmpty() {return c.isEmpty();}
    public boolean contains(Object o) {return c.contains(o);}
    public Object[] toArray() {return c.toArray();}
    public <T> T[] toArray(T[] a) {return c.toArray(a);}
    public String toString() {return c.toString();}
    public boolean add(E e) {throw new UnsupportedOperationException();}
    public boolean remove(Object o) { throw new UnsupportedOperationException();}
    public boolean containsAll(Collection<?> coll) { return c.containsAll(coll);}
    public boolean addAll(Collection<? extends E> coll) { throw new UnsupportedOperationException();}
    public boolean removeAll(Collection<?> coll) { throw new UnsupportedOperationException();}
    public boolean retainAll(Collection<?> coll) { throw new UnsupportedOperationException();}
    public void clear() { throw new UnsupportedOperationException(); }
}
```
The decorator pattern vs. inheritance

• Decorator composes features at run time
  – Inheritance composes features at compile time

• Decorator consists of multiple collaborating objects
  – Inheritance produces a single, clearly-typed object

• Can mix and match multiple decorations
  – Multiple inheritance is conceptually difficult
Today

• More design patterns for reuse
  – Iterator pattern
  – Decorator pattern

• Design goals and design principles
Metrics of software quality, i.e., *design goals*

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>Adherence of implementation to the specifications</td>
</tr>
<tr>
<td>correctness</td>
<td></td>
</tr>
<tr>
<td>Robustness</td>
<td>Ability to handle anomalous events</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Ability to accommodate changes in specifications</td>
</tr>
<tr>
<td>Reusability</td>
<td>Ability to be reused in another application</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Satisfaction of speed and storage requirements</td>
</tr>
<tr>
<td>Scalability</td>
<td>Ability to serve as the basis of a larger version of the application</td>
</tr>
<tr>
<td>Security</td>
<td>Level of consideration of application security</td>
</tr>
</tbody>
</table>

*Source: Braude, Bernstein, Software Engineering. Wiley 2011*
Design principles: heuristics to achieve design goals

- Low coupling
- Low representational gap
- High cohesion
A design principle for reuse: *low coupling*

- Each component should depend on as few other components as possible

- Benefits of low coupling:
  - Enhances understandability
  - Reduces cost of change
  - Eases reuse
Law of Demeter

- "Only talk to your immediate friends"

\[\text{foo.bar()}.\text{baz()}.\text{quz}(42)\]
Representational gap

• Real-world concepts:

• Software concepts:
Representational gap

- Real-world concepts:

- Software concepts:
Representational gap

- Real-world concepts:

- Software concepts:
Benefits of low representational gap

• Facilitates understanding of design and implementation
• Facilitates traceability from problem to solution
• Facilitates evolution
A related design principle: high cohesion

- Each component should have a small set of closely-related responsibilities
- Benefits:
  - Facilitates understandability
  - Facilitates reuse
  - Eases maintenance
Coupling vs. cohesion

• All code in one component?
  – Low cohesion, low coupling

• Every statement / method in a separate component?
  – High cohesion, high coupling
Summary

• Four design patterns to facilitate reuse...

• Design principles are useful heuristics
  – Reduce coupling to increase understandability, reuse
  – Lower representational gap to increase understandability, maintainability
  – Increase cohesion to increase understandability